EG UK Theory and Practice of Computer Graphics (2012)
Hamish Carr and Silvester Czanner (Editors)

ViewFusion: Correlating Structure and Activity Views for Execution Traces

Jonas Trümper[1] Alexandru Telea[2] Jürgen Döllner[1]

[1]Hasso-Plattner-Institute Potsdam, Germany
[2]University of Groningen, the Netherlands

Abstract

Visualization of data on structure and related temporal activity supports the analysis of correlations between the two types of data. This is typically done by linked views. This has shortcomings with respect to efficient space usage and makes mapping the effect of user input into one view into the other view difficult. We propose here a novel, space-efficient technique that 'fuses' the two information spaces – structure and activity – in one view. We base our technique on the idea that user interaction should be simple, yet easy to understand and follow. We apply our technique, implemented in a prototype tool, for the understanding of software engineering datasets, namely static structure and execution traces of the Chromium web browser.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Methodology and Techniques—Interaction techniques

1. Introduction

Structure views are used to display data such as organization layers, software system containment relations, catalogues, and directory structures. Treemaps [SBW08], icicle plots [KL83], and node-link graph layouts [BETT99] are effective tools for creating structure views that support users in tasks such as getting overviews of a given dataset, comparing substructures of interest, and correlating the distribution of metrics of interest with the structure. *Activity views* are equally important: They convey insight into the dynamics of a process, such as the evolution in time of several metrics' values, an event sequence, or the history of an activity. Activity views help reasoning about cause-effect relations, discover trends, and grasp the overall dynamics of a time-dependent process. Activity views involve techniques such as timelines [HHWN02], sequence views [DPH10], flow graphs [San94], and animation [KIL07].

In certain cases, understanding requires combining both structure and activity insights. One such case is *program comprehension* through dynamic analysis in software maintenance [Bas97, WGGS92, LD06]. Here, software is instrumented, and execution data is collected as execution traces (*program tracing*). Use-cases involve understanding a program structure (e.g., its hierarchy of packages, files, and classes) and program execution (e.g., order and duration of function calls). Equally important is the correlation of structure with activity insights to answer questions such as finding high-activity packages; mapping execution phases to program module structures; and reasoning about performance problems at system component level.

In information visualization, many solutions exist for separate visualization of structure and activity. However, combining structure-and-activity data in a single image is still hard. In this paper, we present a new approach for this problem. Rather than using linked views, we 'fuse' both structure and activity information spaces in a single view. Next, we propose several rendering variations and interaction modes that enable users to easily map foci of interest between the two spaces, thereby supporting the task of correlating insights. Our techniques are easy to implement and use, and can be applied to fuse structure and time-dependent data beyond software visualization.

In Section 2, we review related work on structure and activity visualization. Section 3 presents our visual and interaction design. Section 5 presents a program comprehension tool implemented atop of our proposal. Section 6 discusses our techniques. Section 7 concludes the paper.

2. Related Work

In the following, we review related work with a focus on software visualization.

Structure Views: Program structure is typically shown with node-link diagrams using various layout techniques [BETT99, Aub12, AT 10]. For hierarchies (trees), space-filling visualizations such as treemaps [SBW08, vWvdW99], icicle plots [KL83], and radial plots [Hol06] are highly scalable, and can show the correlation of structure with metrics mapped, e.g., to node size and color.

Activity Views: Execution traces are often visualized using different variants of icicle plots. The horizontal axis typically maps time, e.g., function call start and end moments [TBD10] or memory block allocation and release moments [MT07]. The vertical axis maps call stack depth [TBD10] or memory block address ranges [MT07]. Stacked timelines enable comparing the evolution of several time-dependent signals such as software repository commit activities [VTvW05] to find interesting event correlations. Multivariate visualization, e.g., scatter plots and dimensionality reduction techniques, are used to detect correlations in high-dimensional datasets such as multi-metric log files, or to compare different datasets, e.g., profiling data from different execution traces [LPW00, MC01]. Peer-to-peer download metrics [Rob05] and execution traces [VTvW04] are displayed via several Cartesian 2D plots that are further linked by shared axes (dimensions). Various subsampling techniques are used to combine information of subpixel-size events to increase scalability [MT07, CZH*08].

Correlating Views: Structure views are typically used to show the *static* system structure that is obtained, e.g., via static program analysis [NNH05]. In contrast, activity views are used to show *dynamic* information such as execution or software evolution logs. Linked views are probably the most used technique to correlate the two. For example, Cornelissen et al. link a radial bundled node-link view (for static function calls), an icicle plot (for static system structure), and a call timeline (for dynamic execution information) by means of selection and brushing to show which subsystems are active in a certain execution phase [CZH*08]. Similar techniques are used in Tarantula [JH05] and Gammatella [JOH04] to link structure and text views.

Although easy to learn and use, views linked by selection and brushing require a certain effort to use, in particular when the *spatial layout* of the views is different. More precisely, such split-attention setups are known to generate a significant amount of *context switches* that require mental effort, time, and short-term memory to assimilate these distinct views [CKB09]. Especially mentally challenging tasks, such as program comprehension, though, also require users to concentrate and use their short-term memory to correlate pieces of information. Moreover, displaying such views side-by-side, such as when views share an axis, takes a non-negligible amount of screen space. In effect, such separate visualization are not optimal for these tasks. In the following, we present a visualization design that alleviates these problems.

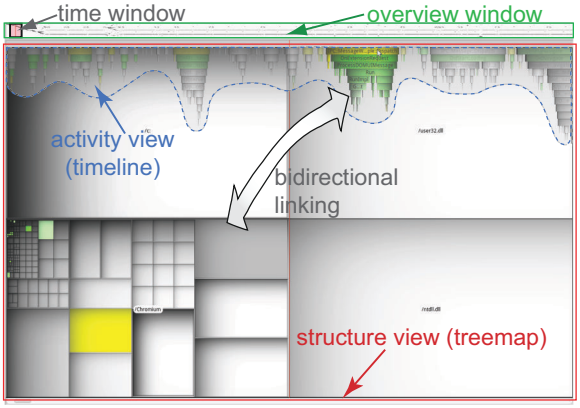

Figure 1: *Combined structure-and-activity view design.*

3. Visual Design

Our goal is to create a visualization design that

- combines structure and activity views in a scalable way;
- enables users to easily correlate subsets of the information shown in the two views;
- is efficient and simple to implement, and portable.

3.1. Overlaid Layout

We start by choosing a treemap and an icicle plot for the structure and activity views, respectively. Our choices are motivated by the high scalability of both views, as shown in numerous cases [vWvdW99, JOH04, CZH*08]. Unlike existing linked view solutions which keep such views spatially separated, we choose to *overlay* the views (Fig. 1).

3.2. Structure View

This view uses a treemap to display a hierarchical (tree) dataset T. In our use-case, T stores containment relationships in a software system. We use here a simple strip layout, which has some advantages as discussed later (Sec. 4.4.1). To maximize information density (one of our design goals), we do not render borders between sibling cells, except a 2-pixel border on the topmost level. Hence, we need other ways to show neighbor cells belonging to the same subtree. An option is to use cushion treemaps (CTMs) [vWvdW99]. CTMs use a Phong-shaded height map built by summing up parabolic profiles $\psi_i : [0,1]^2 \rightarrow \mathbb{R}^+$ centered atop of tree nodes $n_i \in P$ belonging to the same path $P \subset T$. If a slanted light vector is used, shading discontinuities convey the tree-distance of neighbor cells, i.e., the larger the shading discontinuity between those cells, the larger the distance of the respective nodes in T.

The parabolic cushions in the original CTM design have linearly-changing gradients. Thus, to create high-contrast images that convey the tree structure, CTMs use relatively high Phong specular coefficients. This can visibly darken the result (see, e.g., [vWvdW99], Fig. 6). Also, CTMs require per-pixel computations that cannot be efficiently done except if using pixel shaders. Although this is possible [LNVT05], it conflicts with our portability and simplicity requirements.

We take here an approach similar to CTMs, but which is simpler and generates brighter, easier to read, images. For each node $n \in T$ at depth $d_i \geq 0$ from the root of T, we define a *luminance* profile $\psi_i : [0,1]^2 \rightarrow \mathbb{R}^+$ as

$$\psi_i(x,y) = \left[\left(1-(1-f_x x)^{d_i}\right)\left(1-(1-2f_y y)^{d_i}\right)\right]^k, \quad (1)$$

i.e., the product of two exponential profiles ψ_i^x and ψ_i^y (Fig. 2). The values $f_x, f_y \in [0,1]$ control the position of the highlight. Setting $f_x = 1, f_y = 0.8, k = 0.2$ gives an effect similar to the original CTM design.

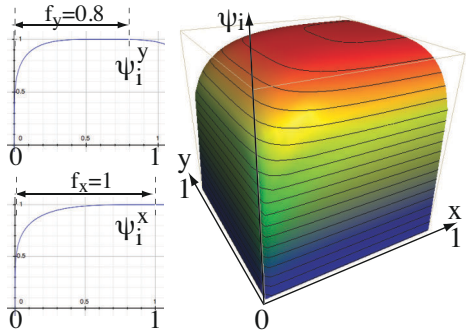

Figure 2: *Treemap cushion luminance design. The 3D plot shows ψ_i for $d_i = 4, f_x = 1, f_y = 0.8$.*

To render our treemap, we multiply, at each screen pixel (x,y), the profiles ψ_i for all treemap cells that cover (x,y). We do this easily by storing ψ_i, for all depths d_i, as 2D luminance textures, and rendering T with textured cells in depth-first order with multiplicative alpha blending. Fig. 3 shows the rendering of the hierarchical structure of a software system with 8,850 elements. Several differences are apparent between our design and CTMs [vWvdW99,LNVT05]. First, the image is much brighter: In contrast to CTMs, the flatness of our profiles ψ_i increases with tree depth, due to the increasing exponent d_i in Eqn. 1. Thus, deep tree cells have a relatively much wider highlight than cells higher in the tree. The asymmetric shading profile, which visually separates neighbor cells whose nodes are far apart in T, is preserved. Overall, our shading slightly reduces the mapping of tree-depth to luminance (present in CTMs) but yields an overall brighter image. As we shall see next in Secs. 4 and 5, this is useful for color-mapping metrics to the structure view.

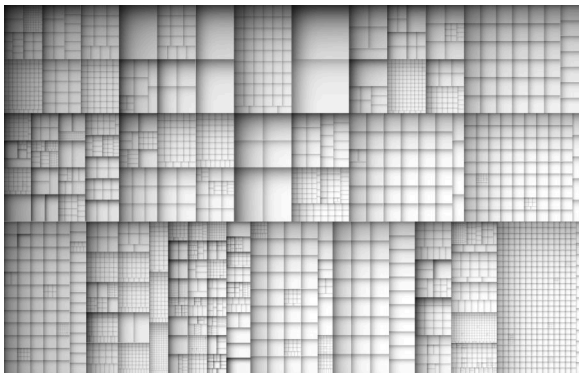

Figure 3: *Treemap rendering for a dataset of 8,850 nodes.*

3.3. Activity View

The activity view displays a sequence $E = \{e_i\}$ of *events*. Each event $e = (e_S, e_E, e_P)$ has a start and end moment $e_S \in \mathbb{R}^+, e_E \in \mathbb{R}^+, e_S < e_E$ and a parent event $e_P \in E \cup NULL$. Thus, E is an event sequence organized as a tree T_E. Parent events encompass time-wise child events, i.e., $\forall e \in E, e_S(e_P(e)) > e_S \wedge e_E(e_P(e)) < e_E$. Such datasets emerge from, e.g., state machine simulations and program tracing. For the latter example, our use-case in this paper, E is a set of function calls, so paths in T_E are program call stacks.

The activity view shows E using an icicle plot metaphor (Fig. 1): Nodes are drawn as rectangular cells in T_E-depth-order from top to bottom, horizontally positioned on their e_S, e_E start and end moments. The activity view can be zoomed and panned with the mouse to focus on a time-range of interest. This is useful when analyzing high-frequency traces that contain many short-duration events.

4. Interaction for Linking and Occlusion Reduction

4.1. Goals

In many applications, like our program comprehension use-case, the structure T in the structure view (Sec. 3.2) and the activity data E in the activity view (Sec. 3.3) are *correlated*: Each event $e \in E$ is associated with a structural element $n \in T$ via an activity-to-structure mapping $m : E \rightarrow T$. Note that m need not be injective. For instance, in our software dataset, m maps from function calls to function declarations; most execution traces have several calls to the same function. Showing such correlations by visualizing m and its inverse m^{-1}, i.e., *linking* the two views, is an important requirement.

A separate issue regards the *occlusion* created by drawing the activity view atop of the structure view (Fig. 1). As noted, we do this to minimize the space needed to show both views. Indeed, if we were to stack the views, this would double the required screen space in the worst case. Overlaying the views is space-efficient, but creates undesired occlusions.

We address both above issues, i.e., visualize the activity-

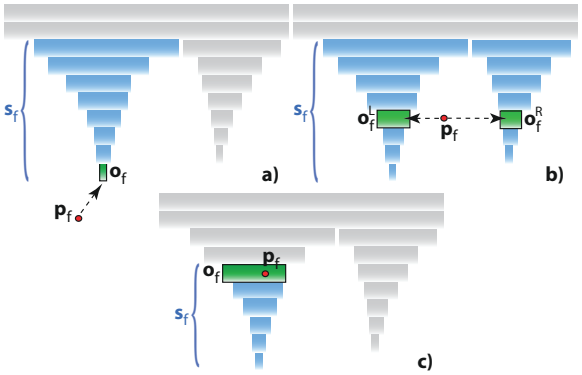

Figure 4: *Interaction modes: Overview (a), approach (b), and detail (c). Focus item is green and focus subset is blue.*

to-structure mapping m and its inverse m^{-1}, and reduce the activity-view *vs* structure-view occlusion, by interaction. To explain this, we first introduce the three key elements of our interaction design: Focus point, focus item, and focus subset. The *focus point* $\mathbf{p}_f = (x_f, y_f) \in \mathbf{R}^2$ is the current mouse position. The *focus item* is the object $o_f \in V$ closest to the focus point in the view $V \in \{T, E\}$ selected for interaction. Users can toggle the view V to interact with, called the *input target*, via the Control key. If $V = T$, $o_f \in T$ is a node in the tree shown in the structure view (Sec. 3.2). If $V = E$, $o_f \in E$ is an event in the sequence shown in the activity view (Sec. 3.3). The *focus subset* $s_f \subset V$ is a set of elements in the view V; s_f contains the focus item o_f and also other objects that are semantically and spatially close to o_f in V.

By suitably choosing o_f and s_f, we address the view-linking and view-occlusion issues, as detailed next.

4.2. Interaction with the Activity View

Interacting with the activity view supports activity-centered use-cases. Following Sec. 4.1, we design suitable definitions of the focus item and focus subset following the visual information-seeking mantra: Overview, zoom-and-filter, and details on demand [Shn96], via three interaction modes.

4.2.1. Overview Mode

In this mode (Fig. 4 a), one typically visualizes a zoomed-out activity view, looking for 'salient' icicles, e.g., deep call stacks surrounded by shallow execution areas. We enter overview mode when the focus point $\mathbf{p}_f$ is outside *and* below the rendered icicle plot. We next define the focus item o_f as the closest (in Euclidean distance sense) icicle plot cell to $\mathbf{p}_f$. Next, we define the focus subset s_f as the path starting at the root of T_E and ending at o_f, whose elements are visible in the activity view at the current zoom and pan levels.

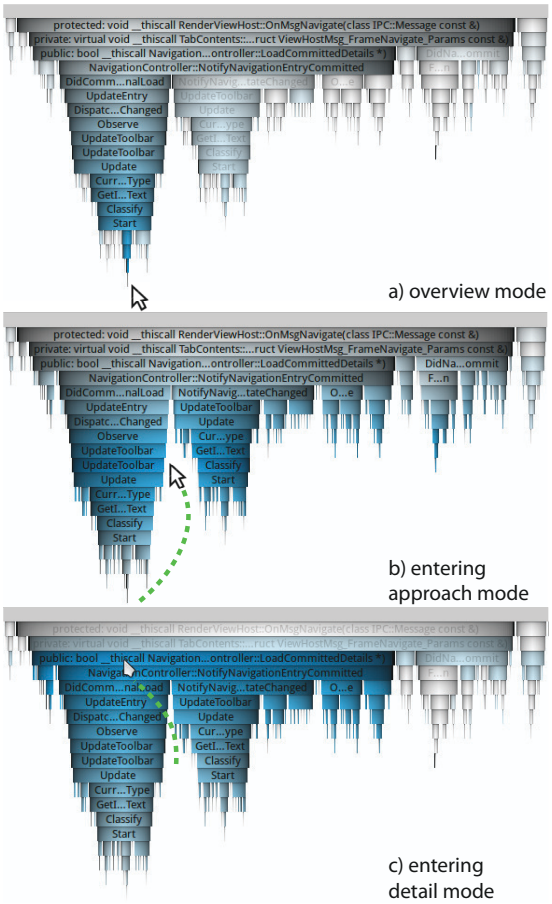

Figure 5: *Interaction modes. Focus color is blue.*

4.2.2. Approach Mode

In this mode (Fig. 4 b), the user moves the mouse closer to the activity view: The focus point $\mathbf{p}_f$ is now *between* two icicles rather than below all icicles as in the overview mode, but still outside the icicle-plot itself. This mode is useful when one has decided to focus on an area within a trace dataset, but is not sure which specific call stacks within that trace deserve further attention. We define *two* focus items o_f^L and o_f^R as the closest items on the x axis to the left, respectively right, of $\mathbf{p}_f$. The focus subset s_f contains now the visible paths in E that pass through o_f^L and o_f^R.

4.2.3. Detail Mode

In this mode (Fig. 4 c), the user moves the mouse, thus $\mathbf{p}_f$, inside the icicle plot, e.g., decides to focus on the call stack below a given function call. We set o_f to the icicle-plot cell under $\mathbf{p}_f$, and s_f to the path from o_f downwards in T_E.

4.2.4. Rendering

Focus items: We render all items in s_f with full opacity and shaded cushions. We use cushions to convey both the

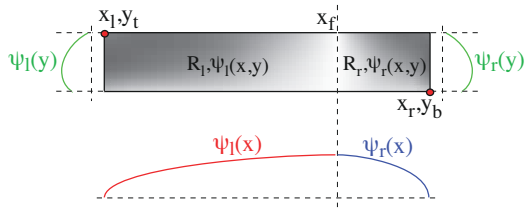

Figure 6: *Shaded cushions for activity view items.*

structure of the selected focus subset s_f and the position of the focus point $\mathbf{p}_f$ within s_f. Consider an item $p \in s_f$ whose icicle-plot cell is a rectangle R spanned by (x_l, y_t) and (x_r, y_b) (Fig. 6). If $x_l < x_f < x_r$, we cut R in two rectangles $R_l = (x_l, y_t); (x_f, y_b)$ and $R_r = (x_f, y_t); (x_r, y_b)$ and texture these with two luminance textures ψ_l and ψ_r, based on Eqn. 1 with $d_i = 1, f_x = 1, f_y = 0.8$. This yields a luminance profile that horizontally varies from dark ($x = x_l$) to fully bright ($x = x_f$) and then to dark again ($x = x_r$), and vertically shows the slightly convex profile in Fig. 2. If $x_f < x_l$ or $x_f > x_r$, we texture R as for the R_r and R_l cases indicated above, respectively. As the user moves the mouse horizontally, the highlight at x_f moves along all items in s_f, like a 3D lighting which glides atop of the focus set.

Items in s_f are further color-mapped to show metrics of interest, as described separately in Sec. 4.4. For any such color mapping, we linearly decrease the saturation of colors in s_f upwards and downwards from o_f until the top-most and bottom-most items in s_f, respectively. As the user moves the mouse vertically within s_f, a saturation highlight follows the mouse to indicate the position of the focus item. Fig. 5 shows the rendering of an activity view with items in s_f colored in blue for illustration purposes. We see how items in the focus set change color close to the mouse. The horizontal shading gradient conveys a soft focus on items close to the mouse, and also emphasizes the icicle plot structure.

Out-of-focus items: All items in $E \setminus s_f$ in the activity view are rendered with a high transparency and low saturation. For example, the right-most call stacks in Fig. 5, are desaturated *and* have a higher luminance (due to the transparent blending on a white background), which allows for visually separating s_f (blue) from its context (gray). This reduces occlusion in two ways. First, one can move the mouse within, or around, the activity view to bring different items in focus, as outlined in Secs. 4.2.1-4.2.3. Secondly, one can grab the activity view and pan it horizontally. The two operations allow for de-occluding any part of the structure view by a mouse gesture (and optional click-to-pan).

4.3. Interaction with the Structure View

As for the activity view, interacting with the structure view requires a focus item o_f and focus subset s_f. The focus point $\mathbf{p}_f$ is always within a treemap cell. We set o_f to this cell, and s_f to the subtree of T containing o_f and starting at a user-specified height h, which is controlled by moving the mouse wheel. Items in $s_f \setminus T$ are rendered with low saturation.

4.4. Color Linking

Colors serve two purposes in our design: First, we *color map* attributes of interest of the items in both the structure and activity views, e.g., package-ID, call stack depth, and function-call starting time. Secondly, we use color to *link* items in focus between the two views, as explained next.

As outlined in Sec. 4.1, we want to bidirectionally link the activity and structure views so one can correlate data shown in both: For an item $u \in E$, we want to show the items $m(u) \subset T$; for an item $v \in T$, we want to show the items $m^{-1}(v) \subset E$. Visualizing m or m^{-1} for *all* items in E and T respectively is hard or even impossible, since T and E may have thousands of items. In our design, this would require, e.g., using a node-link metaphor that connects related items with lines. This can easily lead to unacceptable clutter. Other designs, such as shared view axes [CZH*08,MT07,VTvW04] are not possible given our spatial overlaid design.

To solve this, we restrict ourselves to show m and m^{-1} only for the focus subset s_f. For this, we propose two color-linking designs. In the first design, called *data-in-focus*, items in s_f are colored via a task-specific colormap. Fig. 8 shows this for calls in $s_f \subset E$ colored by relative stack depth. Corresponding structure items $\{m(u) | u \in s_f\} \subset T$ use the same colormap. This shows how our metric (call stack depth) for the selected calls (s_f) varies over the function definitions ($m(s_f)$). Items $u \in E \setminus s_f$ are drawn in both views with no color mapping, i.e., gray. As the user changes s_f by brushing over E, the colored items change in both views, which allows for linking subsets of interest in these views.

In our second design, called *data-outside-focus*, items in s_f and $m(s_f)$ are left gray, and items in $E \setminus s_f$ and $T \setminus m(s_f)$ are color mapped. As the user changes s_f by brushing over E, linked items appear as gray items in both views. In contrast to the first color linking design, we can now use *different* color mappings in the two views, e.g., to show call count in the activity view and call duration in the structure view (Fig. 9), since linking is shown by the common gray color. This mode supports the task of identifying linked items in both views, shown in the context of view-specific metrics mapped in each view by view-specific color maps.

Color linking works for selections s_f done in both the activity and structure views. In other words, we can either select items in the activity view and see where they map in the structure view (m mapping), or select items in the structure view and see where these map in the activity view (m^{-1} mapping). As both views are drawn in the same screen rectangle, we toggle the input target view by pressing the Control key, as outlined in Sec. 4.1.

4.4.1. Constrained Structure-View Layout

We can further exploit the treemap layout to minimize structure-view *vs* activity-view occlusion: We define a function $\gamma : T \to \mathbb{N}$ equal to the number of relations of a node $n \in T$ to the event sequence E, i.e., $\gamma = \|m^{-1}\|$. When using the strip treemap layout, we sort nodes in T on γ. Treemap cells that have many relations to the activity view, e.g., often-called functions, are placed at the treemap bottom, while items with few relations go to the top. This reduces the likelihood of occlusion between both views during color linking.

5. Applications

We have implemented the proposed visualization atop of a toolset for program-execution comprehension [TBD10]. As input data, we recorded an execution of Chromium, the open-source code base of Google's web browser Chrome [Goo12]. The hierarchical structure contains 8,850 files and folders, in total 2.7 million lines of C/C++ code. The trace, pre-filtered as the system was instrumented only partially, results in about 9,000 calls to 914 function bodies.

Within all color mappings presented next, we use two special colors to indicate missing data: Items in the focus subset which have no data are *white*; items outside the focus subset which have no data are *blue*. We next present several analyses centered on correlating software structure with trace data, implemented with our proposed techniques.

5.1. Temporal Locality and Coherence of Packages

As a first use-case, we analyze how distinct packages of Chromium collaborate in time (Fig. 7). We use a categorical color mapping for both views (data-in-focus color linking, Sec. 4.4). Colors map the package-ID for a few top-level folders of interest in the structure tree T. Next, we move the mouse to select the topmost node in the activity view, i.e., focus on the entire visible trace. We see that the yellow and green packages are executed at the beginning of the shown time frame. In contrast, the red, cyan and blue packages are involved over the entire time frame.

5.2. Structural Locality and Coherence of Events and Packages

We now analyze structural properties of events shown in the activity view. Given a focus subset $s_f \subset E$ in the event view, we first partition s_f into subsets of events $s_f^j = m^{-1}(f_j)$, where $\{f_j\} = \{m(e) | e \in s_f\}$ are all files in T related to events in s_f. For each file f_j, we then define two metrics m_a (activity view) and m_s (structure view) based on the events in s_f^j. Both metrics are normalized to $[0,1]$ based on their global minimum and maximum values for all events in s_f. The metrics are then color-mapped using a 'criticality' scheme (green=low, yellow=middle, red=high).

5.2.1. Stack Depth: Analyzing Per-File Coherence

In execution trace analysis, stack depth is an important measure: It gives a first hint of whether a specific function implements high-level or low-level functionality, also measured as utilityhood [HL05]. Typically, low stack depth means low utilityhood and high stack depth means high utilityhood. While this metric is known to work quite well at function level, we want to see how it behaves at file level, given that files can contain multi-level functionality.

To analyze coherence of utilityhood in the trace, we set $m_a(f_j) \equiv m_s(f_j)$ to the maximum call-stack depth of all correlated events in s_f^j. We visualize these metrics using data-in-focus color-linking and the criticality color scheme ((Fig. 8). In the activity view, we can see that the file containing *TabContents::NotifyNavigationStateChanged* (1 in Fig. 8) is, despite its low 'visual' stack depth, colored red instead of the expected green. This means that this file not only contributes to high-level, but also to low-level functionality. In contrast, most other calls in the activity view show a green-to-red downward gradient, i.e. they are defined in files whose overall functionality is homogeneous. In the structure view, we see a rather heterogeneous distribution of functionality levels in the *chrome* package.

5.2.2. Call Count and Duration: High-Activity Packages

Call count and call duration are often used to find how actively packages take part in the execution of specific functionality. To show these, we define $m_a = \|s_f^j\|$, i.e., the number of calls to f_j (call count per file), and $m_s = \sum_{e \in s_f^j} (e_E - e_S)$, i.e., cumulated call duration of f_j. Since $m_a \neq m_s$, we use the data-outside-focus color linking design (Sec. 4.4).

When brushing the structure view, we first see that most treemap cells are blue (Fig. 9). As this color indicates missing (trace) data (Sec. 5), this shows that our recorded execution trace 'samples' the system structure only sparsely.

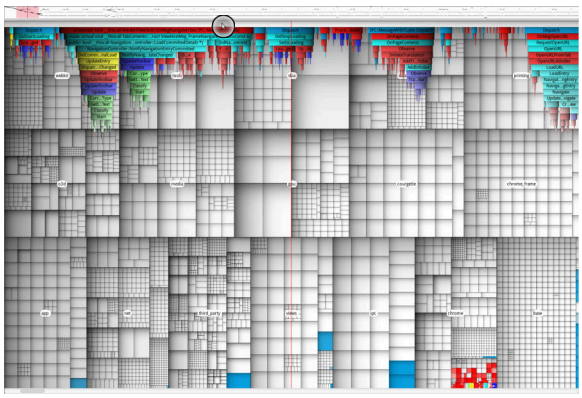

Figure 7: *Colors: Package ID. Input target: Activity view, Interaction mode: Detail.*

This is indeed so given our partial instrumentation of the code stack. This is a quick way for users to assess the overall code coverage of a given trace. Next, we see that the color distribution in the activity view is heavily shifted to low values (green and yellow). This is due to an outlier that is visible with the naked eye in the activity view (zoom-inset 1 in Fig. 9): This is a timer function, *ResetBaseTimer*, called every few milliseconds and thus having a very high call count. In contrast, for the call duration metric shown in the structure view, we see no such outlier (no red treemap cells): Called functions have similar durations. Further on, in the *chrome* package, which contains the current focus subset s_f brushed by the user (2 in Fig. 9), several files are correlated to events in the activity view. In contrast, in most other packages only few files are correlated to events in the activity view. This is a further indication that the trace examined here 'targets' mainly functionality in the *chrome* package.

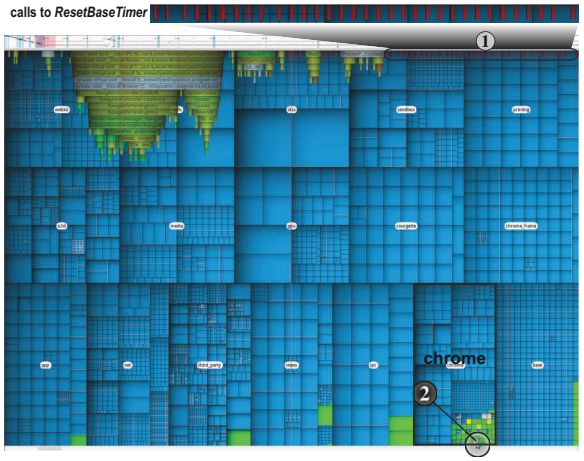

Figure 9: *Colors: Call count (in activity view) and call duration (in structure view). Input target: Structure view.*

6. Discussion

Generality: Our approach can show a hierarchy and an event-sequence, and highlight relationships between items in the two datasets. Although we used our approach on program static structure and execution traces, the proposal is in no way restricted to software engineering datasets; they generally apply to structural data and correlated temporal data.

Scalability: Given the space-filling treemap layout and the space-efficient icicle plot layout, the design scales well to datasets containing thousands of elements. The shaded cushions used in both views convey additional structural cues, e.g., nesting of events in the activity view and hierarchy in the structure view. This makes the views usable in zoomed-out mode when items are only a few pixels large.

Ease of Use: The proposed interaction techniques are easy to use: Selecting item subsets of interest in either of the views, and zooming and panning to a time-range of interest in the activity view, are done using only the mouse motion, mouse wheel, and the Control key. Although we have not conducted a formal user evaluation, insight so far shows that the proposed method is intuitive and easy to use within minutes. We expect that integrating multitouch-based input devices allows for further increasing the easiness of user input.

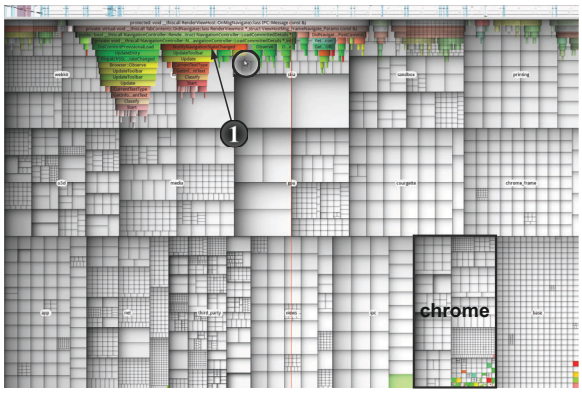

Figure 8: *Colors: Maximum stack depth. Input target: Activity view, Interaction mode: Approach.*

Flexibility: The proposed brushing-based selection techniques allow for easily and quickly adjusting the level-of-detail of the selection by moving the mouse or turning the mouse wheel. Our two color-linking techniques (data-in-focus, data-outside-focus) based on the selection help correlating two *different* metrics at a time. Also, the linking techniques enable us to project any metric computed on one of the two datasets to the other dataset, effectively enabling a 'push'/'pull' of metric data from one view to the other.

Implementation: A simple and portable implementation is a key requirement for the acceptance of software visualization tools [Kie06, Kos03]. Our implementation, done in Qt, requires only basic texture-mapping and alpha blending, and renders datasets of tens of thousands of elements in real-time, including the interactive brushing.

Limitations: Occlusion reduction by pan-and-brush combined with transparency (Sec. 4.2.4) works well even for large datasets, e.g., deep icicle plots overlaid on deep treemaps - one can 'see through' any icicle plot area just by moving the mouse and/or panning this plot. However, this still requires a (small) amount of user interaction. The likelihood of occlusion is further reduced by our strip treemap reordering based on relation count. Even more aggressive occlusion reduction could be done by exploiting treemap layouts which allow for more flexible reordering schemes.

The color-linking technique (Sec. 4.4) can only show relationships between *groups* of items, i.e., our focus subset s_f *vs* its mapping $m(s_f)$. Although one-to-one relations can be inferred by seeing how highlighted elements

change while brushing and/or selecting smaller subsets s_f, this does not replace a detailed relationship visualization. Further refinements could add, e.g., carefully routed bundled edges [Hol06] atop of our design to emphasize such details.

7. Conclusions

We have presented a method for the visualization of combined hierarchical structure and event-sequence datasets. We address visual scalability by a new overlaid layout of icicle plots and treemaps. By this, the two views are linked with less disruption. We use simple, brushing-based, interaction for selection of items of interest and occlusion reduction. Combined with color-linking, this allows for querying relationships between parts of the two displayed datasets. Computationally-efficient shaded cushion variations are proposed for structure and focus enhancement. The technique is illustrated on large datasets from program comprehension, but can be used on other structure-and-activity datasets.

Acknowledgements

This work was funded by the Federal Ministry of Education and Research (BMBF), Germany within the InnoProfile Transfer research group "4DnD-Vis".

References

[AT 10] AT & T: The Graphviz package, 2010. `www.graphviz.org`. 2

[Aub12] AUBER D.: Tulip visualization system. `tulip.labri.fr`. 2

[Bas97] BASILI V. R.: Evolving and Packaging Reading Technologies. *JSS 38*, 1 (1997), 3–12. 1

[BETT99] BATTISTA G. D., EADES P., TAMASSIA R., TOLLIS I. G.: *Graph Drawing: Algorithms for the visualization of graphs*. Prentice Hall, 1999. 1, 2

[CKB09] COCKBURN A., KARLSON A., BEDERSON B. B.: A review of overview+detail, zooming, and focus+context interfaces. *ACM Computing Surveys. 41*, 1 (Jan. 2009), 2:1–2:31. 2

[CZH*08] CORNELISSEN B., ZAIDMAN A., HOLTEN D., MOONEN L., VAN DEURSEN A., VAN WIJK J. J.: Execution Trace Analysis through Massive Sequence and Circular Bundle Views. In *J. Sys. & Software* (2008), Elsevier, (Ed.), vol. 81, pp. 2252–2268. 2, 5

[DPH10] DE PAUW W., HEISIG S.: Visual and Algorithmic Tooling for System Trace Analysis: A Case Study. *SIGOPS Oper Syst Rev 44*, 1 (2010), 97–102. 1

[Goo12] GOOGLE INC.: Google Chrome browser, 2012. `www.google.com/chrome`. 6

[HHWN02] HAVRE S., HETZLER E., WHITNEY P., NOWELL L.: ThemeRiver: Visualizing Thematic Changes in Large Document Collections. *IEEE TVCG 8* (2002), 9–20. 1

[HL05] HAMOU-LHADJ A.: *Techniques to Simplify the Analysis of Execution Traces for Program Comprehension*. PhD thesis, University of Ottawa, 2005. 6

[Hol06] HOLTEN D.: Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. In *Proc. IEEE InfoVis* (2006), pp. 741–748. 2, 8

[JH05] JONES J., HARROLD M. J.: Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proc. ASE* (2005), pp. 237–243. 2

[JOH04] JONES J. A., ORSO A., HARROLD M. J.: GAMMATELLA: Visualizing Program-Execution Data for Deployed Software. *Palgrave Macmillan Inf. Vis. 3*, 3 (Sept. 2004), 173–188. 2

[Kie06] KIENLE H.: *Building Reverse Engineering Tools with Software Components*. PhD thesis, Univ. of Victoria, Canada, 2006. 7

[KIL07] KADABA N. R., IRANI P. P., LEBOE J.: Visualizing Causal Semantics Using Animations. In *IEEE Comput. Graph. Appl.* (2007). 1

[KL83] KRUSKAL J. B., LANDWEHR J. M.: Icicle Plots: Better Displays for Hierarchical Clustering. *American Statistician 37*, 2 (1983), 162–168. 1, 2

[Kos03] KOSCHKE R.: Software Visualization in Software Maintenance, Reverse Engineering, and Re-Engineering: A Research Survey. *J. Soft. Maint. and Evol. 15*, 2 (2003), 87–109. 7

[LD06] LATOZA T., DELINE G. V. R.: Maintaining Mental Models: A Study of Developer Work Habits. In *Proc. ICSE* (2006), pp. 492–501. 1

[LNVT05] LOMMERSE G., NOSSIN F., VOINEA L., TELEA A.: The *Visual Code Navigator*: An Interactive Toolset for Source Code Investigation. In *Proc. Infovis* (2005), IEEE, pp. 24–31. 3

[LPW00] LEON D., PODGURSKI A., WHITE L. J.: Multivariate Visualization in Observation-Based Testing. In *Proc. ICSE* (2000), pp. 116–125. 2

[MC01] MOC J., CARR D.: Understanding Distributed Systems via Execution Trace Data. In *Proc. IWPC* (2001), pp. 60–67. 2

[MT07] MORETA S., TELEA A.: Multiscale Visualization of Dynamic Software Logs. In *Proc. EuroVis* (2007), pp. 11–18. 2, 5

[NNH05] NIELSON F., NIELSON H. R., HANKIN C.: *Principles of Program Analysis*. Springer, 2005. 2

[Rob05] ROBERTS J.: TraceVis: An Execution Trace Visualization Tool. In *Proc. MoDS* (2005), pp. 123–130. 2

[San94] SANDER G.: Graph Layout through the VCG Tool. In *Proc. Graph Drawing* (1994), Springer, pp. 194–205. 1

[SBW08] SHNEIDERMAN B., BEDERSON B., WATTENBERG M.: The Treemap 4.0 Visualization System, 2008. http://www.cs.umd.edu/hcil/treemap. 1, 2

[Shn96] SHNEIDERMAN B.: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proc. IEEE Symp. on Visual Languages* (1996), pp. 336–343. 4

[TBD10] TRÜMPER J., BOHNET J., DÖLLNER J.: Understanding Complex Multithreaded Software Systems by Using Trace Visualization. In *Proc. SoftVis* (2010), ACM, pp. 133–142. 2, 6

[VTvW04] VOINEA L., TELEA A., VAN WIJK J. J.: EZEL: a Visual Tool for Performance Assessment of Peer-to-Peer File-Sharing Networks. In *Proc. InfoVis* (2004), pp. 41–48. 2, 5

[VTvW05] VOINEA L., TELEA A., VAN WIJK J. J.: CVSscan: visualization of code evolution. In *Proc. SoftVis* (2005), ACM, pp. 47–56. 2

[vWvdW99] VAN WIJK J. J., VAN DE WETERING H.: Cushion Treemaps: Visualization of Hierarchical Information. In *Proc. InfoVis* (1999), pp. 73–78. 2, 3

[WGGS92] WILDE N., GOMEZ J. A., GUST T., STRASBURG D.: Locating User Functionality in Old Code. In *Proc. ICSM* (1992), pp. 200–205. 1